UNIT I – FOUNDATIONS OF HIGH PERFORMANCE COMPUTING

3. Memory Hierarchy & Cache Optimization

Data can be stored in a computer system in many different ways. CPUs have a set of registers, which can be accessed without delay. In addition there are one or more small but very fast caches holding copies of recently used data items. Main memory is much slower, but also much larger than cache. Finally, data can be stored on disk and copied to main memory as needed. This a is a complex hierarchy, and it is vital to understand how data transfer works between the different levels in order to identify performance bottlenecks.

3.1. Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die. The need for caches can be easily understood by realizing that data transfer rates to main memory are painfully slow compared to the CPU's arithmetic performance.

Need for Cache Memory:

While peak performance soars at several GFlops/sec per core, memory bandwidth, i.e., the rate at which data can be transferred from memory to the CPU, is still stuck at a couple of GBytes/sec, which is entirely insufficient to feed all arithmetic units and keep them busy continuously. To make matters worse, in order to transfer a single data item (usually one or two DP words) from memory, an initial waiting time called latency passes until data can actually flow. Thus, latency is often defined as the time it takes to transfer a zero-byte message. Memory latency is usually of the order of several hun dred CPU cycles and is composed of different contributions from memory chips, the chipset and the processor. Although Moore's Law still guarantees a constant rate of improvement in chip complexity and (hopefully) performance, advances in memory performance show up at a much slower rate. The term DRAM gap has been coined for the increasing "distance" between CPU and memory in terms of latency and bandwidth. Caches can alleviate the effects of the DRAM gap in many cases.

3.2. The Memory Hierarchy

Usually there are at least two levels of cache called L1 and L2, respectively. L1 cache is split into two parts — one for instructions (L1I) and one for data (L1D). Outer levels (L2, L3) are unified, storing both. The closer a cache is to the CPU registers,

the lower its latency and higher its bandwidth — but also the smaller it must be to keep administrative overhead manageable.

Cache Hit and Cache Miss

When CPU issue a load instruction for a data item, the L1 cache logic checks whether that item already resides in cache.

- (Cache Hit) If data is found in cache → served immediately at low latency → fast execution continues
- (Cache Miss) If data is not in cache → must be fetched from outer cache levels or, worst case, from main memory → CPU stalls and waits

Types of Cache Misses

Cache misses occur when the processor attempts to access data that is not present in the cache. When this happens, the data needs to be fetched from the slower main memory, which results in increased latency. There are four primary types of cache misses: compulsory misses, capacity misses, conflict misses, and coherence misses. Understanding these misses is crucial for optimizing memory access performance.

1. Compulsory Misses

Compulsory misses occur when a memory block is accessed for the first time, meaning it's not yet in the cache. Since the data has never been loaded before, the cache will miss the access and need to fetch the data from the main memory.

These misses are inevitable and cannot be avoided, especially for new data accesses. However, they can be minimized by increasing cache efficiency or using better prefetching techniques.

2. Capacity Misses

Description: Capacity misses happen when the cache does not have enough space to store all the data needed by the processor. When the cache becomes full, older or unused data may be evicted, resulting in misses when the data is accessed again.

These misses are more likely to occur when dealing with large datasets or applications that require more data than the cache can hold.

3. Conflict Misses

Conflict misses occur in direct-mapped and set-associative caches due to the limited number of locations available for storing data. If multiple memory blocks map to the same cache location, one block will evict another, causing a miss when the evicted block is accessed again.

Conflict misses are more common in direct-mapped caches, where each memory block is mapped to a specific location

4. Coherence Misses

Coherence misses, also known as invalidation misses, occur in systems with multiple processors or cores. These misses arise when a processor's cache holds stale data that has been modified by another processor or an external device (e.g., I/O devices). These misses typically occur when data is invalidated or modified in one cache, and the change must be reflected in other caches. Coherence misses can introduce delays due to the need for data synchronization across processors or devices.

3.3 Cache Mapping

Cache mapping is the method used to decide **which cache line stores a particular block of main memory**. Without a mapping rule, the processor would not know where to search for data in the cache. There are three major cache mapping techniques:

1. Fully Associative Mapping
2. Direct Mapping
3. Set-Associative Mapping

Fully Associative Mapping

Any memory block can be placed in **any cache line**, giving maximum flexibility and minimizing conflict misses. However, the hardware must search all cache lines to locate data and use a replacement policy (such as LRU) when the cache is full, making it expensive and complex.

Direct-Mapped Cache

Each memory block maps to exactly one predetermined cache line, making cache lookup very fast and simple. However, if multiple frequently used memory blocks

map to the same cache line, they repeatedly replace each other, causing cache thrashing and conflict misses.

Set-Associative Cache

The cache is divided into sets, and each memory block can be placed in any one of the cache lines (ways) within its assigned set. This provides a balance between speed and flexibility by greatly reducing conflict misses while keeping hardware complexity manageable.

3.3. Cache Optimization Techniques

Optimization of cache performance ensures that it is utilized in a very efficient manner to its full potential. Methods for reducing Hit Time, Miss Rate, and Miss Penalty:

Methods to reduce Hit Time:

- **Small and simple caches** If lesser hardware is required for the implementation of caches, then it decreases the Hit time
- **Avoid Address translation during indexing**: Address translation can be avoided by using virtual caches. Hence, they help in reducing the hit time.

Methods to reduce Miss Rate:

- **Larger block size**: Larger block size implies a lesser number of blocks which results in increased conflict misses.
- **Larger cache size:** Increasing the cache size results in a decrease of capacity misses, thereby decreasing the miss rate.
- **Higher associativity:** Higher associativity results in a decrease in conflict misses. Thereby, it helps in reducing the miss rate.

Methods to reduce Miss Penalty:

- **Multi-Level Caches:** If there is only one level of cache, then we need to decide between keeping the cache size small in order to reduce the hit time or making it larger so that the miss rate can be reduced.
- Both of them can be achieved simultaneously by introducing cache at the next levels.

Temporal Locality

Data that was recently used is likely to be used again soon. If a variable is loaded into cache and reused several times before being evicted, that is good temporal locality. The cache reuse ratio β captures this — the fraction of memory accesses satisfied from cache

Spatial Locality

Data stored near recently accessed data is likely to be accessed soon. This is exploited through **cache lines**. All data transfers between cache and main memory happen in fixed-size units called cache lines (typically 64–128 bytes, or 8–16 double-precision words). When a cache miss occurs on one element, the entire cache line containing that element is fetched from memory.

3.4. Prefetching

Prefetching is a cache optimization technique used to improve memory access performance by loading data into the processor's cache before it is explicitly requested. By predicting future memory accesses, prefetching reduces memory access latency and the effective cache miss penalty. In modern processors, prefetching is typically performed at the cache-line level, where one or more contiguous cache lines are fetched into the cache in advance. This approach exploits spatial locality, as programs often access neighboring memory locations sequentially.

There are several methods for implementing prefetching:

Hardware Prefetching: Modern processors include dedicated hardware units that automatically monitor memory access patterns and predict which cache lines are likely to be accessed next. These cache lines are fetched into the cache in advance without requiring software intervention.

Software Prefetching: Programmers explicitly insert prefetch instructions into the code to request that specific memory locations be loaded into the cache before they are accessed. This approach requires knowledge of the application's memory access patterns and the underlying hardware.

Compiler-Assisted Prefetching: Some compilers can analyze memory access patterns and automatically insert prefetch instructions during code generation. This reduces the programming effort while improving cache performance.